PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

Recomputation based Implementations of And-Or Parallel Prolog

Gopal Gupta†
Department of Computer Science
Box 30001, Dept. 3CU,
New Mexico State University
Las Cruces, NM 88003-0001
gupta@nmsu.edu

Manuel V. Hermenegildo
Facultad de Informática
Universidad Politécnica de Madrid
28660-Boadilla del Monte, Madrid, SPAIN
herme@fi.upm.es

Abstract

We argue that in order to exploit both Independent And- and Or-parallelism in Prolog programs there is advantage in recomputing some of the independent goals, as opposed to all their solutions being reused. We present an abstract model, called the Composition-Tree, for representing and-or parallelism in Prolog Programs. The Composition-tree closely mirrors sequential Prolog execution by recomputing some independent goals rather than fully re-using them. We also outline two environment representation techniques for And-Or parallel execution of *full* Prolog based on the Composition-tree model abstraction. We argue that these techniques have advantages over earlier proposals for exploiting and-or parallelism in Prolog.

1. Introduction

One of the features of logic programming languages that make them attractive is that they allow implicit parallel execution of programs. There are three main forms of parallelism present in logic programs: or-parallelism, Independent And-parallelism and Dependent and-parallelism. In this paper we restrict ourselves to Or-parallelism and Independent and-parallelism. There have been numerous proposals for exploiting or-parallelism in logic programs [AK90, HC87, LW90, W84, W87, etc.]‡ and quite a few for exploiting independent and-parallelism [H86, LK88, etc.]. Models have also been proposed to exploit both or-parallelism and independent and-parallelism in a single framework [BK88, GJ89, RK89]. It is the latter aspect of combining independent and- and or-parallelism that this paper addresses.

One aspect which most models that have been proposed (and some implemented) so far for combining or-parallelism and independent and-parallelism have in common is that they have either considered only pure logic programs (pure Prolog), e.g. [RK89, GJ89], or, alternatively, modified the language to separate parts of the program that contain extra-logical predicates (such as cuts and side-effects) from those that contain purely logical predicates, then allowing parallel execution only in parts containing purely logical predicates [RS87, BK88]. In the former case practical Prolog programs cannot be executed since most such programs use extra-logical features. The latter approach has a number of disadvantages: first, it requires programmers to divide the program into sequential and parallel parts themselves. As a result of this, parallelism is not exploited completely implicitly since some programmer intervention is required. This also rules out the possibility of taking "dusty decks" of existing Prolog programs and running them in parallel. In addition, some parallelism may also be lost since parts of the program that contain side-effects may also actually be the parts that contain parallelism. It has been shown that or-parallelism and independent and-parallelism can be exploited in *full* Prolog completely implicitly (for example, in the Aurora and Muse Systems [HC88, LWH90, AK91], and in the &-Prolog system [HG90, MH89, CC89]). We argue that the same can be done for systems that combine independent and- and or-parallelism and that will be one of the design objectives of the approach presented in this paper.†

The paper thus describes a general approach for

† Much of this work was done while the first author was a Research Associate in David H.D. Warren's group at the University of Bristol.

‡ See [GJ90] for a systematic analysis of the various models.

† Due to length limitations the actual techniques for incorporating side effects in and-or parallel systems in order to execute full Prolog are presented in a separate report [GS91]. However, the model presented in this paper has been designed with this issue in mind, i.e., having as one of the objectives that the inclusion of side effects be facilitated.

combined exploitation of independent and- and or-parallelism in full Prolog. We present an abstract model of and-or parallelism for logic programs which mirrors sequential Prolog execution more closely, essentially by recomputing some independent goals (those that Prolog recomputes) rather than re-using them, and show the advantages of this approach. Our presentation is then two-pronged, in that we propose two alternative efficient environment representation techniques to support the model: paged binding arrays and stack copying. Using the concept of teams of processors‡, we also briefly discuss issues such as scheduling and memory management.

The environment representation techniques proposed are extensions of techniques designed for purely or-parallel systems—specifically the Aurora [LW90] and Muse [AK90] systems. The method for encoding independent and-parallelism is taken from purely independent and-parallel systems—specifically the &-Prolog system [HG90]: we use the parallel conjunction operator "&" to signify parallel execution of the goals separated by this operator and Conditional Graph Expressions (CGEs) [HN86,H86]§. Hence our model can be viewed as a combination of the &-Prolog system and a purely or-parallel system such as Aurora or Muse—in the presence of only independent and-parallelism our model behaves *exactly* like &-Prolog while in the presence of only or-parallelism it behaves *exactly* like the Aurora or Muse systems, depending on the environment representation technique chosen.

The rest of the paper is organised as follows: Section 2 describes or-parallelism and independent and-parallelism in Prolog programs. Section 3 presents arguments for favouring recomputation of some independent and-parallel goals over their complete reuse. Section 4 then presents an abstract model called the Composition-tree for representing and-or parallel execution of Prolog with recomputation. Section 5 deals with environment representation issues in the Composition-tree: section 5.1 presents a comparison of environment representation techniques based on whether there is *sharing* or *non-sharing*; section 5.2 presents an extension of the Binding Arrays method, an environment representation technique based on sharing; while section 5.3 presents another technique, based on non-sharing, which employs *stack-copying*. Finally, section 6 presents our conclusions. We assume that the reader is familiar to some extent with Binding Arrays [W84, W87], the Aurora and Muse Systems [LWH90, AK90], and the &-Prolog system [HG90], as well as with some aspects of sequential Prolog implementation.

‡ We refer to the working "agents" of the system –the "workers" of Aurora and Muse and "agents" of &-Prolog– simply as processors, under the assumption that the term will generally represent processes mapped onto actual processors in an actual implementation.

§ Note that CGEs and & operators can be introduced automatically in the program at compile time [MH89a] using abstract interpretation and thus the programmer is not burdened with the parallelization task.

2. Or- and Independent And-parallelism

Or-parallelism arises when more than one rule defines some relation and a procedure call unifies with more than one rule head in that relation—the corresponding bodies can then be executed in or-parallel fashion. Or-parallelism is thus a way of efficiently searching for solutions to a goal, by exploring alternative solutions in parallel. It corresponds to the parallel exploration of the branches of the proof tree. Or-parallelism has successfully been exploited in full Prolog in the Aurora [LWH90] and the Muse [AK90] systems both of which have shown very good speed up results over a range of problems.

Informally, *Independent And-parallelism* arises when more than one goal is present in the query or in the body of a procedure, and the run-time bindings for the variables in these goals are such that two or more goals are *independent* of one another. In general, independent and-parallelism includes the parallel execution of any set of goals in a resolvent, provided they meet some independence condition. Independent and-parallelism is thus a way of speeding up a problem by executing its subproblems in parallel. One way for goals to be independent is that they don't share any variable at run-time (*strict* independence [HR90]†). This can be ensured by checking that their resulting argument terms after applying the bindings of the variables are either variable-free (i.e., *ground*) or have non-intersecting sets of variables. Independent and-parallelism has been successfully exploited in the &-Prolog system [HG90]. Independent and-parallelism is expressed in the &-Prolog system through the parallel conjunction operator "&", which will also be used in this paper. For syntactic brevity we will also use &-Prolog's Conditional Graph Expressions (CGEs), which are of the form

$$(condition \ \Rightarrow \ goal_1 \ \& \ goal_2 \ \& \ \ldots \ \& \ goal_n \ )$$

meaning, using the standard Prolog if-then-else construct,

$$(condition \ \rightarrow \ goal_1 \ \& \ldots \& \ goal_n \ ; \ goal_1, \ldots, \ goal_n)$$

† There is a more general concept of independence, *non-strict* independence [HR90], for which the same results (the model presented in this paper included) apply. However, the rest of the presentation in this section will refer for simplicity, and without loss of generality, to strict independence.

i.e., that, if *condition* is true, goals $goal_1 \ldots goal_n$ are to be evaluated in parallel, otherwise they are to be evaluated sequentially. The *condition* can obviously be any prolog goal but is normally a conjunction of special builtins which include *ground*/1, which checks whether its argument has become a ground term at run-time, or *independent*/2, which checks whether its two arguments are such at run-time that they don't have any variable in common, or the constant *true* meaning that $goal_1 \ldots goal_n$ can be evaluated in parallel unconditionally. It is possible to generate parallel conjunctions and or CGEs automatically and quite successfully at compile-time using abstract interpretation [MH89]. Thus, exploitation of independent and-parallelism in &-Prolog is completely implicit (although user annotation is also allowed).

There have been a number of attempts to exploit or- and independent and-parallelism together in a single framework [GJ89, RK89, WR87, etc.], however, and as mentioned earlier, they either don't support the full Prolog language, or require user intervention. Also, in general these systems advocate *solution sharing* which, as will be argued in the following section, stands in the way of supporting full Prolog.

3. Recomputation *vs* Reuse

In the presence of both and- and or-parallelism in logic programs, it is possible to avoid recomputing certain goals. This has been termed as solution sharing [GJ89, G91a]. For example, consider two independent goals `a(X), b(Y)`, each of which has multiple solutions. Assuming that all solutions to the program are desired, the most efficient way to execute this goal would be to execute `a` and `b` in their entirety and combine their solutions (possibly incrementally) through a join [BK88, GJ89, RK89]. However, to solve the above goal in this way one needs to be sure that the set of solutions for `a` and `b` are *static* (i.e., if either goal is executed multiple times, then each invocation produces an identical set of solutions). Unfortunately, this can hold true only if clauses for `a` and `b` are pure logic programs. If side-effects are present (as is usually the case with Prolog programs), then the set of solutions for these goals may not be static. For example, consider the case where, within `b`, the value of a variable is read from the standard input and then some action taken which depends on the value read. The solutions for `b` may be different for every invocation of `b` (where each invocation corresponds to a different solution of `a`), even if the goal is completely independent of the others. Hence solution sharing would yield wrong results in such a case. The simple solution of sequentializing such and-parallel computations results in loss of too much and-parallelism, because if `a(X), b(Y)` falls in the scope of some other goal, which is being executed in and-parallel, then that goal has to be sequentialized too, and we have to carry on this sequentialization process right up to the top level query. If, however, the goals are recomputed then this sequentialization can be avoided, and parallelism exploited even in the presence of cuts and side-effects [GS91].

Hence, there is a strong argument for recomputing non-deterministic and-parallel goals, especially, if they are not pure, and even more so if we want to support Prolog as the user language†. Additionally, recent simulations of and-or parallelism [SH91] show that typical Prolog programs perform very little recomputation, thus providing further evidence that the amount of computation saved by a system which avoids recomputation may be quite small in practice. Presumably this behaviour is due to the fact that Prolog programmers, aware of the selection and computation rules of Prolog, order literals in ways which result in efficient search which minimises the recomputation of goals. Note that the use of full or partial recomputation can never produce any slowdown with respect to Prolog since Prolog itself uses full recomputation.

Recomputation of independent goals was first proposed in the context of &-Prolog.‡. It is obviously also used in Aurora and Muse (since, performing no goal independence analysis, no possibility of sharing arises) and has made these three systems quite capable of supporting full Prolog. Recomputation in the context of and-or parallelism has also been proposed in [SH91]§. The argument there was basically one of ease of simulation and, it was argued, of implementation (being a simulation study no precise implementation approach was given). Here we add the important argument of being able to support full Prolog, provide an abstract representation of the corresponding execution tree, and outline two efficient implementation approaches.

4. And-Or Composition Tree

The most common way to express and- and or-

† There is a third possibility as well: to recompute those independent and-parallel goals that have side-effects and share those that don't. Since the techniques for implementing solution sharing are in the literature and techniques for implementing solution recomputation are presented herein such an approach would represent a –perhaps non-trivial– combination of the given methods.

‡ In the case of &-Prolog there are even further arguments in favour of recomputation, related to management of a single binding environment and memory economy.

§ The idea of recomputation is referred to as "or-under-and" in [SH91].

parallelism in logic programs is through the traditional concept of and-or trees, i.e. trees consisting of or-nodes and and-nodes. Or-nodes represent multiple clause heads matching a goal while and-nodes represent multiple subgoals in the body of a clause being executed in and-parallel. Since in the model presented herein we are representing and-parallelism via parallel conjunctions, our and-nodes will represent such conjunctions. Thus, given a clause `q :- (true => a & b)`, and assuming that a and b have 3 solutions each (to be executed in or-parallel form) and the query is `?- q`, then the corresponding and-or tree would appear as shown in figure 1.

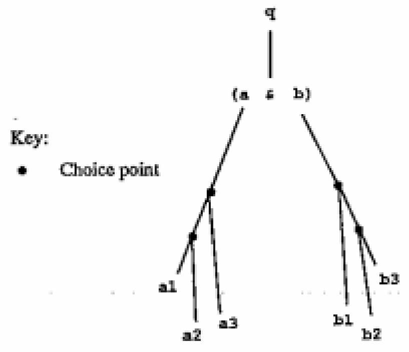


Figure 1: And-Or Tree

One problem with such a traditional and-or tree is that bindings made by different alternatives of a are not visible to different alternatives of b, and vice-versa, and hence the correct environment has to be created before the continuation goal of the parallel conjunction can be executed. Creation of the proper environments requires a global operation, for example, *Binding Array loading* in AO-WAM [GJ89, G91a], the complex dereferencing scheme of PEPSys [BK88], or the "global forking" operation of the Extended Andorra Model [W90]. To eliminate this possible source of overhead in our model, we extend the traditional and-or tree so that the various or-parallel environments that simultaneously exist are always separate.

The extension essentially uses the idea of recomputing independent goals of a parallel conjunction of &-Prolog [HG90] (and Prolog!). Thus, for every alternative of a, the goal b is computed in its entirety. Each separate combination of a and b is represented by what we term as a *composition* node (c-node for brevity). Thus, each composition node in the tree corresponds to a different solution for the parallel conjunction, i.e., a different "continuation". Thus the extended tree, called the *Composition-tree* (C-tree for brevity), for the above query might appear as shown in figure 2—for each alternative of the and-parallel goal a, goal b is entirely recomputed (in fact, the tree could contain up to 9 c-nodes, one for each combination of solutions of a and b). To represent the fact that a parallel conjunction can have multiple solutions we add a branch point (choice point) before the different composition nodes. Note that c-nodes and branch points serve purposes very similar to the Parcall frames and markers of the RAP-WAM [H86, HG90]. The C-tree can represent or- and independent and-parallelism quite naturally—execution of goals in a c-node gives rise to independent and-parallelism while parallel execution of untried alternatives gives rise to or-parallelism.†.

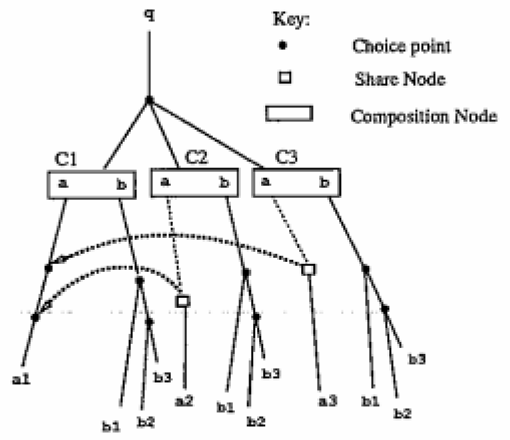


Figure 2: Composition Tree

Notice the topological similarity of the C-tree with the purely or-parallel tree shown in figure 3 for the program above. Essentially, branches that are "shared" in the purely or-parallel tree (i.e. that are "common", even though different binding environments may still have to be maintained –we will refer to such branches and regions for simplicity simply as "shared") are also shared in the C-tree. This sharing is represented by means of a *share-node*, which has a pointer to the shared branch and a pointer to the composition node where that branch is needed (figure 2). Due to sharing the subtrees of some independent and-parallel goals maybe spread out across different composition nodes. Thus, the subtree of goal a is spread out over c-nodes C1, C2 and C3 in the C-tree of figure 2, the total amount of program-related work being essentially maintained.

† In fact, a graphical tool capable of representing this tree has shown itself to be quite useful for implementors and users of independent and- and or-parallel systems [CG91].

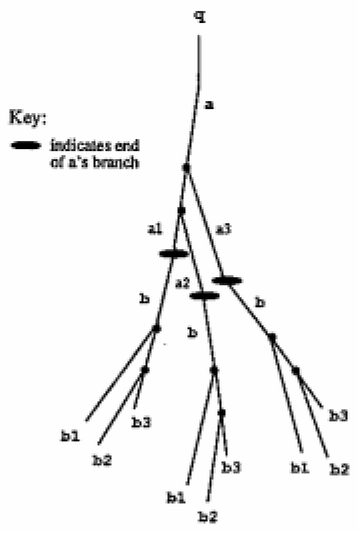


Figure 3: Or-Parallel Tree

4.1 And-Or Parallelism & Teams of Processors

We will present some of the implementation isuues from the point of view of extending an or-parallel system to support independent and-parallelism. When a purely or-parallel model is extended to exploit independent and-parallelism then the following problem arises: at the end of independent and-parallel computation, all participating processors should see all the bindings created by each other. However, this is completely opposite to what is needed for or-parallelism where processors working in or-parallel should not see the (conditional) bindings created by each other. Thus, the requirements of or-parallelism and independent and-parallelism seem anti-thetical to each other. The solutions that have been proposed range from updating the environment at the time independent and-parallel computations are combined [RK89, GJ89] to having a complex dereferencing scheme [BK88]. All of these operations have their cost.

We contend that this cost can be eliminated by organising the processors into teams. Independent and-parallelism is exploited among processors within a team while or-parallelism is exploited among teams. Thus a processor within a team would behave like a processor in a purely and-parallel system while all the processors in a given team would collectively behave like a processor in a purely or-parallel system. This entails that all processors within each team share the data structures that are used to maintain the separate or-parallel environments. For example, if binding arrays are being used to represent multiple or-parallel environments, then only one binding array should exist per team, so that the whole environment is visible to each member processor of the team. In copying is used, then processors in the team share the copy. Note that in the limit case there will be only one processor per team. Also note that despite the team arrangement a processor is free to migrate to another team as long as it is not the only one left in the team. Although a fixed assignment of processors to teams is possible a flexible scheme appears preferable. This will be discussed in more detail in section 4.3. The concept of teams of processors has been successfully used in the Andorra-I system [SW91], which extends an or-parallel system to accommodate dependent and-parallelism.

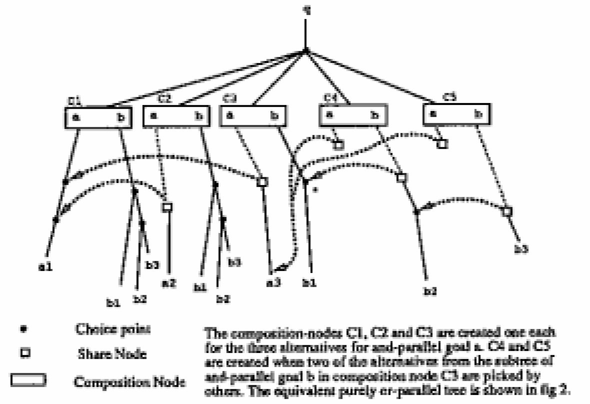
4.2. C-tree & And-Or Parallelism

The concept of organising processors into teams also meshes very well with C-trees. A team can work on a c-node in the C-tree—each of its member processors working on one of the independent and-parallel goal in that c-node. We illustrate this by means of an example. Consider the query corresponding to the and-or tree of figure 1. Suppose we have 6 processors P1, P2, ..., P6, grouped into 3 teams of 2 processors each. Let us suppose P1 and P2 are in team 1, P3 and P4 in team 2, and P5 and P6 in team 3. We illustrate how the C-tree shown in figure 2 would be created.

Execution commences by processor P1 of team 1 picking up the query q and executing it. Execution continues like normal sequential execution until the parallel conjunction is encountered, at which point a choice point node is created to keep track of the information about the different solutions that the parallel conjunction might generate. A c-node is then created (node C1 in figure 2). The parallel conjunction consists of two and-parallel goals a and b, of which a is picked up by processor P1, while b is made available for and-parallel execution. The goal b is subsequently picked up by processor P2, teammate of processor P1. Processor P1 and P2 execute the parallel conjunction in and-parallel producing solutions a1 and b1 respectively. In the process they leave choice points behind. Since we allow or-parallelism below and-parallel goals, these untried alternatives can be processed in or-parallel by other teams. Thus the second team, consisting of P3 and P4 picks up the untried alternative corresponding to a2, and the third team, consisting of P5 and P6, picks up the untried alternative corresponding to a3. Both these teams create a new c-node, and restart the execution of and-parallel goal b (the goal to the right of goal a): the first processor in each team (P3 and P5, respectively) executes the alternative for a, while the second processor in each team (P4 and P6, respectively) executes the restarted goal b. Thus, there are

3 copies of b executing, one for each alternative of a. Note that the nodes in the subtree of a, between c-node C1 and the choice points from where untried alternatives were picked, are "shared" among different teams (in the same sense as the nodes above the parallel conjunction are—different binding environments still have to be maintained).

Since there are only three teams, the untried alternatives of b have to be executed by backtracking. In the C-tree, backtracking always takes place from the right to mimic Prolog's behaviour—goals to the right are completely explored before a processor can backtrack inside a goal to the left. Thus, if we had only one team with 2 processors, then only one composition node would actually need to be created, and all solutions would be found via backtracking, exactly as in &-Prolog, where only one copy of the Parcall frame exists [H86, HG90]. On the other hand if we had 5 teams of 2 processors each, then the C-tree could appear as shown in fig 4. In figure 4, the 2 extra teams steal the untried alternatives of goal b in c-node C3, This results in 2 new c-nodes being created, C4 and C5 and the subtree of goal b in c-node C3 being spread across c-nodes C3, C4 and C5. The topologically equivalent purely or-parallel tree of this C-tree is still the one shown in figure 3. The most important point to note is that new c-nodes get created only if there are resources to execute that c-node in parallel. Thus, the number of c-nodes in a C-tree can vary depending on the availability of processors.



The composition-nodes C1, C2 and C3 are created one each for the three alternatives for and-parallel goal a. C4 and C5 are created when two of the alternatives from the subtree of and-parallel goal b in composition node C3 are picked by others. The equivalent purely or-parallel tree is shown in fig 2.

Figure 4: C-tree for 5 Teams

It might appear that intelligent backtracking, that accompanies independent and-parallelism in &-Prolog, is absent in our abstract and-or parallel C-tree model. This is because if b were to completely fail, then this failure will be replicated in each of the three copies of b. We can incorporate intelligent backtracking by stipulating that an untried alternative be stolen from a choice point, which falls in the scope of a parallel conjunction, only after at least one solution has been found for each goal in that parallel conjunction. Thus, c-nodes C2, C3, C4 and C5 (fig 4) will be created only after the first team (consisting of P1 and P2) succeeds in finding solutions a1 and b1 respectively. In this situation if b were to fail, then the c-node C1 will fail, resulting in the failure of the whole parallel conjunction.

4.3. Processor Scheduling

Since our abstract model of C-trees is dependent upon the number of processors available, some of the processor scheduling issues can be determined at an abstract level, without going into the details of a concrete realization of the C-trees. As mentioned earlier, teams of processors are used to carry out or-parallel work while individual processors within a team perform and-parallel work. Since and-parallel work is shared within a team, a processor can in principle steal and-parallel work only from members of its own team. Or-parallel work is shared at the level of teams, thus only an idle team can steal an untried alternative from a choice point. An idle processor will first look for and-parallel work in its own team. If no and-parallel work is found, it can decide to migrate to another team where there is work, provided it is not the last remaining processor in that team. If no such team exists it can start a new team of its own, perhaps with idle processors of other teams, and the new team can steal or-parallel work. One has to carefully balance the number of teams and the number of processors in each team, to fully exploit all the and- and or-parallelism available in a given Prolog program†.

5. Environment Representation

So far we have described and-or parallel execution with recomputation at an abstract level. We have not addressed the crucial problem of environment representation in the C-tree. In this section we discuss how to extend the Binding Arrays (BA) method [W84,W87] and the Stack-copying [AK90] methods to solve this problem. These extensions enable a team of processors to share a single BA without wasting too much space.

5.1 Sharing *vs* Non-Sharing

In an earlier paper [GJ90] we argued that environment representation schemes that have constant-time task creation and constant-time access to variables, but non-constant time task-switching, are superior to those

† Some of the 'flexible scheduling' techniques that have been developed for the Andorra-I system [D91] can be directly adapted for optimal distribution of or- and and-parallel work.

methods which have non-constant time task creation or non-constant time variable-access. The reason being that the number of task-creation operations and the number of variable-access operations are dependent on the program, while the number of task-switches can be controlled by the implementor by carefully designing the work-scheduler.

The schemes that have constant-time task creation and variable-access can be further subdivided into those that physically share the execution tree, such as Binding Arrays scheme [W84, W87, LW90] and Versions Vectors [HC87] scheme, and those that do not, such as MUSE [AK90] and Delphi [CA88]. Both these kinds of schemes have their advantages. The advantage of non-sharing schemes such as Muse and Delphi are that less synchronization is needed in general since each processor has its own copy of the tree and thus there is less parallel overhead [AK90]. This also means that they can be implemented on non-shared memory machines more efficiently. However, operations that may require synchronization and voluntary suspension such as side effects, cuts and speculative scheduling are more overhead prone to implement. When an or-parallel system reaches a side effect which is in a non-leftmost or-branch, it has two choices: (i) it can suspend the current branch and switch to some other node where there is work available, the suspended branch would be woken up when it becomes leftmost; or (ii) it can busy-wait at the current branch until it becomes left most. In case (i) an or-parallel system that does not share the execution tree, such as Muse, will have to save its current execution stack in a scratch memory-area since switching to a new node means that the current stack would be overwritten due to copying of the branches corresponding to the new node. Even if modern sophisticated multiprocessor Operating Systems may allow some memory-saving optimizations, a substantial memory overhead may still be present†. The same holds for case (ii), where a modern OS may manage to avoid busy-waiting, but at the cost of extra memory.

The essential conclusion is that for some applications (those that require processors to synchronize often due to presence of a large number of side-effects and cuts) environment representation schemes which share the or-tree are better, and for some other applications (those that require processors to synchronize less often) schemes which maintain an independent or-tree per processor are better. With this observation in mind we have extended both types of environment representation schemes to accommodate independent and-parallelism with recomputation of goals. We first describe an extension of the Binding Arrays scheme, and then an extension of the stack-copying technique. Due to space limitations the essence of both approaches will be presented rather than specifying them in detail as full models, which is left as future work.

† Experimental results show that processors may voluntarily suspend as much as 10 to 100s of times for large sized programs [SI91].

5.2. Environment Representation using BAs

Recall that in the binding-array method [W84, W87] an offset-counter is maintained for each branch of the or-parallel tree for assigning offsets to conditional variables (CVs)† that arise in that branch. The 2 main properties of the BA method for or-parallelism are the following:

(i) The offset of a conditional variable is fixed for its entire life.

(ii) The offsets of two consecutive conditional variables in an or-branch are also consecutive.

The implication of these two properties is that conditional variables get allocated space consecutively in the binding array of a given processor, resulting in optimum space usage in the BA. This is important because a large number of conditional variables might need to be created at runtime‡.

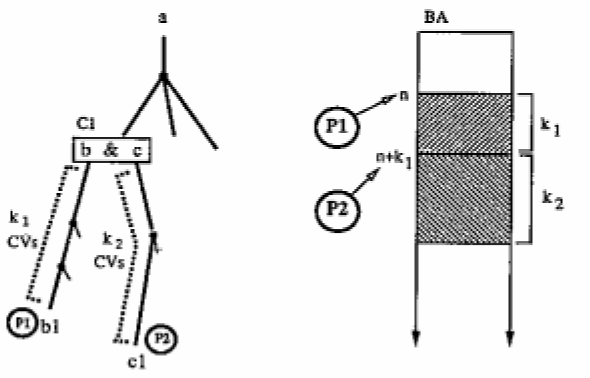


Fig (i): Part of a C-tree   Figure (ii): Optimal Space Allocation in the BA

Figure 5: BAs and Independent And-Parallelism

In the presence of independent and-parallel goals, each of which has multiple solutions, maintaining contiguity in the BA can be a problem, especially if processors are allowed (via backtracking or or-parallelism) to search for these multiple solutions. Consider a goal with a parallel conjunction: `a, (true => b & c), d.` A part of its C-tree is shown in figure 5(i) (the figure

† Conditional variables are variables that receive different bindings in different environments [GJ90].

‡ For instance, in Aurora [LW90] about 1Mb of space is allocated for each BA.

also shows the number of conditional variables that are created in different parts of the tree). If b and c are executed in independent and-parallel by two different processors P1 and P2, then assuming that both have private binding arrays of their own, all the conditional variables created in branch b–b1 would be allocated space in BA of P1 and those created in branch of c–c1 would be allocated space in BA of P2. Likewise conditional bindings created in b would be recorded in BA of P1 and those in c would be recorded in BA of P2. Before P1 or P2 can continue with d after finding solutions b1 and c1, their binding arrays will have to be merged somehow. In the AO-WAM [GJ89, G91a] the approach taken was that one of P1 or P2 would execute d after updating its Binding Array with conditional bindings made in the other branch (known as the the BA loading operation). The problem with the BA loading operation is that it acts as a sequential bottleneck which can delay the execution of d, and reduce speedups. To get rid of the BA loading overhead we can have a common binding array for P1 and P2, so that once P1 and P2 finish execution of b and c, one of them immediately begins execution of d since all conditional bindings needed would already be there in the common BA. This is consistent with our discussion in section 4.1 about having teams of processors where all processors in a team would share a common binding array.

However, if processors in a team share a binding array, then backtracking can cause inefficient usage of space, because it can create large unused holes in the BA. This is because processors in a team, that are working on different independent and-parallel branches, will allocate offsets in the binding array concurrently. The exact number of offsets needed by each branch cannot be allocated in advance in the binding array because the number of conditional variables that will arise in a branch cannot be determined a priori. Thus, the offsets of independent and-branches will overlap: for example, the offsets of k_1 CVs in branch b1 will be intermingled with those of k_2 CVs in branch c1. Due to overlapping offsets, recovery of these offsets, when a processor backtracks, requires tremendous book-keeping. Alternatively, if no book-keeping is done, it leads to large amount of wasted space that becomes unusable for subsequent offsets (see [GS92, G91, G91a] for more details).

5.2.1. Paged Binding Array

To solve the above problem we divide the binding array into *fixed sized segments*. Each conditional variable is bound to a pair consisting of a segment number and an offset within the segment. An auxiliary array keeps track of the mapping between the segment number and its starting location in the binding array. Dereferencing CVs now involves double indirection: given a conditional variable bound to $\langle i, o \rangle$, the starting address of its segment in the BA is first found from location i of the auxiliary array, and then the value at offset o from that address is accessed. A set of CVs that have been allocated space in the same logical segment (i.e. CVs which have common i) can reside in any physical page in the BA, as long as the starting address of that physical page is recorded in the ith slot in the auxiliary array. Note the similarity of this scheme to memory management using paging in Operating Systems, hence the name Paged Binding Array (PBA)†. Thus a segment is identical to a page and the auxiliary array is essentially the same as a page table. The auxiliary and the binding array are common to all the processors in a team. From now on we will refer to the BA as the Paged Binding Array (PBA), the auxiliary array as the Page Table (PT), and our model of and-or parallel execution as the PBA model‡.

Every time execution of an and-parallel goal in a parallel conjunction is started by a processor, or the current page in the PBA being used by that processor for allocating CVs becomes full, a *page-marker node* containing a unique integer id i is pushed onto the trail-stack. The unique integer id is obtained from a shared counter (called a pt_counter). There is one such counter per team. A new page is requested from the PBA, and the starting address of the new page is recorded in the ith location of the Page Table. i is referred to as the page number of the new page. Each processor in a team maintains an offset-counter, which is used to assign offsets to CVs within a page. When a new page is obtained by a processor, the offset-counter is reset. Conditional variables are bound to the pair <i, o>, where i is the page number, and o is the value of the offset-counter, which indicates the offset at which the value of the CV would be recorded in the page. Every time a conditional variable is bound to such a pair, the offset counter o is incremented. If the value of o becomes greater than K, the fixed page size, a new page is requested and new page-marker node is pushed.

† Thanks to David H. D. Warren for pointing out this similarity.

‡ A paged binding array has also been used in the ElipSys system of ECRC [VX91], but for entirely different reasons. In ElipSys, when a choice point is reached the BA is replicated for each new branch. To reduce the overhead of replication, the BA is paged. Pages of the BA are copied in the children branches on demand, by using a "copy-on-write" strategy. In ElipSys, unlike our model, paging is not necessitated by independent and-parallelism.

A list of free pages in the PBA is maintained separately (as a linked list). When a new page is requested, the page at the head of the list is returned. When a page is freed by a processor, it is inserted in the free-list. The free-list is kept ordered so that pages higher up in the PBA occur before those that are lower down. This way it is always guaranteed that space at the top of the PBA would be used first, resulting in optimum space usage of space in the PBA.

While selecting or-parallel work, if the untried alternative that is selected is not in the scope of any parallel conjunction, then task-switching is more or less like in purely or-parallel system (such as Aurora), modulo allocation/deallocation of pages in the PBA. If, however, the untried alternative that is selected is in the and-parallel goal g of a parallel conjunction, then the team updates its PBA with all the conditional bindings created in the branches corresponding to goals which are to the left of g. Conditional bindings created in g above the choice point are also installed. Goals to the right of g are restarted and made available to other member processors in the team for and-parallel execution. Notice that if a C-tree is folded into an or-parallel tree according to the relationship shown in figures 2 and 3, then the behaviour of (and the number of conditional bindings installed/deinstalled during) task switching would closely follow that of a purely or-parallel system such as Aurora, if the same scheduling order is followed.

Note that the paged binding array technique is a generalization of the environment representation technique of AO-WAM [GJ89, G91a], hence some of the optimizations [GJ90a] developed for the AO-WAM, to reduce the number of conditional bindings to installed/deinstalled during task-switching, will also apply to the PBA model. Lastly, seniority of conditional variables, which needs to be known so that "older" variables never point to "younger ones", can be easily determined with the help of the $<i, o>$ pair. Older variables will have a smaller value of i; and if i is the same, then a smaller value of o.

More details on Paged Binding Arrays can be found in [GS92, G91].

5.3. The Stack Copying Approach

An alternative approach to represent multiple environments in the C-tree is to use explicit *stack-copying*. Rather than sharing parts of the tree, the shared branches can be explicitly copied, using techniques similar to those employed by the MUSE system [AK90].

To briefly summarize the MUSE approach, whenever a processor P1 wants to share work with another processor P2 it selects an untried alternative from one of the choice points in P2's stack. It then copies the entire stack of P2, backtracks up to that choice point to undo all the conditional bindings made below that choice point, and then continues with the execution of the untried alternative. In this approach, provided there is a mechanism for copying stacks, the only cells that need to be shared during execution are those corresponding to the choice points. Execution is otherwise completely independent (modulo side-effect synchronization) in each branch and identical to sequential execution.

If we consider the presence of and-parallelism in addition to or-parallelism, then, depending on the actual types of parallelism appearing in the program and the nesting relation between them, a number of relevant cases can be distinguished. The simplest two cases are of course those where the execution is purely or-parallel or purely and-parallel. Trivially, in these situations standard MUSE and &-Prolog execution respectively applies, modulo the memory management issues, which will be dealt with in section 5.3.2.

Of the cases when both and- and or-parallelism are present in the execution, the simpler one represents executions where and-parallelism appears "under" or-parallelism but not conversely (i.e. no or-parallelism appears below c-nodes). In this case, and again modulo memory management issues, or-parallel execution can still continue as in Muse while and-parallel execution can continue like &-Prolog (or in any other local way. The only or-parallel branches which can be picked up appear then above any and-parallel node in the tree. The process of picking up such branches would be identical to that described above for MUSE.

In the presence of or-parallelism under and-parallelism the situation becomes slightly more complicated. In that case, an important issue is carefully deciding which portions of the stacks to copy. When an untried alternative is picked from a choice-point, the portions that are copied are precisely those that have been labelled as "shared" in the C-tree. Note that these will be precisely those branches that will also be copied in an equivalent (purely or-parallel) MUSE execution. In addition, precisely those branches will be recomputed that are also recomputed in an equivalent (purely and-parallel) &-Prolog execution.

Consider the case when a processor selects an untried alternative from a choice point created during execution of a goal g_j in the body of a goal which occurs after a parallel conjunction where there has been and-parallelism above the the selected alternative, but all the forks are finished. Then not only will it have to copy all the stack segments in the branch from the root to the parallel conjunction, but also the portions of stacks corresponding to all the forks inside the parallel conjunction and those of the goals between the end of the parallel conjunction and g_j. All these segments have in principle to be copied because the untried alternative may have access to variables in all of them and may modify such variables.

On the other hand, if a processor selects an untried alternative from a choice point created during execution of a goal g_i inside a parallel conjunction, then it will have to copy all the stack segments in the branch from the root to the parallel conjunction, and it will also have to copy the stack segments corresponding to the goals $g_1 \ldots g_{i-1}$ (i.e. goals to the left). The stack segments up to the parallel conjunction need to be copied because each different alternative within the g_is might produce a different binding for a variable, `X`, defined in an ancestor goal of the parallel conjunction. The stack segments corresponding to goals g_1 through g_{i-1} have to be copied because the different alternatives for the goals following the parallel conjunction might bind a variable defined in one of the goals $g_1 \ldots g_{i-1}$ differently.

5.3.1. Execution with Stack Copying

We now illustrate by means of a simple example how or-parallelism can be exploited in non deterministic and-parallel goals through stack copying. Consider the tree shown in figure 1 that is generated as a result of executing a query `q` containing the parallel conjunction `(true => a(X) & b(Y))`. For the purpose of illustration we assume that there is an unbounded number of processors, P1 ... Pn.

Execution begins with processor P1 executing the top level query q. When it encounters the parallel conjunction, it picks the subgoal a for execution, leaving `b` for some other processor. Let's assume that Processor P2 picks up goal `b` for execution (figure 6.(i)). As execution continues P1 finds solution `a1` for a, generating 2 choice points along the way. Likewise, P2 finds solution `b1` for `b`.

Since we also allow for full or-parallelism within and-parallel goals, a processor can steal the untried alternative in the choice point created during execution of `a` by P1. Let us assume that processor P3 steals this alternative, and sets itself up for executing it. To do so it copies the stack of processor P1 up to the choice point (the copied part of the stack is shown by the dotted line; see index at the bottom of figure 6), simulates failure to remove conditional bindings made below the choice point, and restarts the goals to its right (i.e. the goal `b`). Processor P4 picks up the restarted goal `b` and finds a solution `b1` for it. In the meantime, P3 finds the solution `a2` for a (see figure 6.(ii)). Note that before P3 can commence with the execution of the untried alternative and P4 can execute the restarted goal `b`, they have to make sure that any conditional bindings made by P2 while executing `b` have also been removed. This is done by P3 (or P4) getting a copy of the trail stack of P2 and resetting all the variables that appear in it.

Like processor P3, processor P5 steals the untried alternative from the second choice point for a, copies the stack from P1 and restarts `b`, which is picked up by processor P6. As in MUSE, the actual choice point frame is shared to prevent the untried alternative in the second choice point from being executed twice (once through P1 and once through P3). Eventually, P5 finds the solution `a3` for `a` and P6 finds the solution `b1` for `b`.

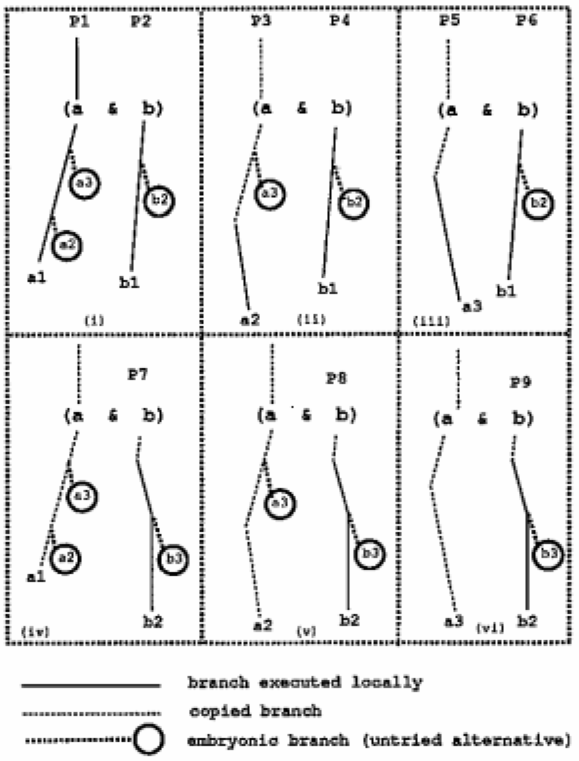


Figure 6: Parallel Execution with Stack Copying

Note that now 3 copies of `b` are being executed, one for each solution of a. The process of finding the solution `b1` for `b` leaves a choice point behind. The untried alternative in this choice point can be picked up for execution by another processor. This is indeed what is done by processors P7, P8 and P9 for each copy of `b` that is executing. These processors copy the stack of P2, P4 and P6, respectively, up to the choice point.

The stack segments corresponding to goal a are also copied (figures 6.(iv), 6.(v), 6.(vi)) from processors P1, P3 and P5, respectively. The processors P7, P8 and P9 then proceed to find the solution b2 for b.

Execution of the alternative corresponding to the solution b2 in the three copies of b produces another choice-point. The untried alternatives from these choice points can be picked up by other idle teams in a manner similar to that for the previous alternative of b (not shown in figure 6). Note that if there were no processors available to steal the alternative (corresponding to solution b3) from b then this solution would have been found by processors P7, P8 and P9 (in the respective copies of b that they are executing) through backtracking as in &-Prolog. The same would apply if no processors were available to steal the alternative from b corresponding to solution b2.

5.3.2. Managing the Address Space

While copying stack segments we have to make sure that pointers in copied portions do not need relocation. In Muse this is ensured by having a physically separate but logically identical memory spaces for each of the processors [AK90]. In the presence of and-parallelism and teams of processors a more sophisticated approach has to be taken.

All processors in a team share the same logical address space. If there are n processors in the team the address space is divided up into m memory segments ($m \geq n$). The memory segments are numbered from 1 to m. Each processor allocates its heap, local stacks, trail etc. in one of the segments (this also implies that the maximum no. of processors that a team can have is m). Each team has its own independent logical address space, identical to the address space of all other teams. Also, each team has an identical number of segments. Processors are allowed to switch teams so long as there is a memory segment available for them to allocate their stacks in the address space of the other team.

Consider the scenario where a choice point, which is not in the scope of any parallel conjunction, is picked up by a team Tq from the execution tree of another team Tp. Let x be the memory segment number in which this choice point lies. The root of the Prolog execution tree must also lie in memory segment x since the stacks of a processor cannot extend into another memory segment in the address space. Tq will copy the stack from the xth memory segment of Tp into its own xth memory segment. Since the logical address space of each team is identical and is divided into identical segments, no pointer relocation would be needed. Failure is then simulated and the execution of the untried alternative of the stolen choice point begun. In fact, the copying of stacks can be done incrementally as in MUSE [AK90] (other optimizations in MUSE to save copying should apply equally well to our model, and are left as future work).

Now consider the more interesting scenario where a choice point, which lies within the scope of a parallel conjunction, is picked up by a processor in a team Tq from another team Tp. Let this parallel conjunction be the CGE ($true \Rightarrow g_1 \& \ldots \& g_n$) and let g_i be the goal in the parallel conjunction whose sub-tree contains the stolen choice point. Tq needs to copy the stack segments corresponding to the computation from the root up to the parallel conjunction and the stack segments corresponding to the goals g_1 through g_i. Let us assume these stack segments lie in memory segments of team Tp and are numbered $x_1, \ldots, x_k$. They will be copied into the memory segments numbered $x_1, \ldots, x_k$ of team Tq. Again, this copying can be incremental. Failure would then be simulated on g_i. We also need to remove the conditional bindings made during the execution of the goal $g_{i+1} \ldots g_n$ by team Tp. Let $x_{k+1} \ldots x_l$ be the memory segments where $g_{i+1} \ldots g_n$ are executing in team Tp. We copy the trail stacks of these segments and reinitialize (i.e. mark unbound) all variables that appear in them. The copied trail stacks can then be discarded. Once removal of conditional bindings is done the execution of the untried alternative of the stolen choice point is begun. The execution of the goals $g_{i+1} \ldots g_n$ is restarted and these can be executed by other processors which are members of the team. Note that the copied stack segments occupy the same memory segments as the original stack segments. The restarted goals can however be executed in any of the memory segments.

An elaborate description of the stack-copying approach, with techniques for supporting side-effects, various optimizations that can be performed to improve efficiency, and implementation details are left as future work. Preliminary details can be found in [GH91].

6. Conclusions & Comparison with Other Work

In this paper, we presented a high-level approach capable of exploiting both independent and-parallelism and or-parallelism in an efficient way. In order to find all solutions to a conjunction of non-deterministic and-parallel goals in our approach some goals are explicitly recomputed as in Prolog. This is unlike in other and-or parallel systems where such goals are shared. This allows our scheme to incorporate side-effects and to support Prolog as the user language more easily and simplifies other implementation issues.

In the context of this approach we also presented two techniques for environment representation in the presence of independent and-parallelism which are extensions of highly successful environment representation techniques for supporting or-parallelism. The first technique, based on Binding Arrays [W84, W87], and termed Paged Binding Array technique, yields a system which can be viewed as a direct combination of the Aurora [LW90] and &-Prolog [HG90] systems. The second technique based on stack copying [AK90] yields a system which can be viewed as a direct combination of the MUSE [AK90] and &-Prolog systems. If an input program has only or-parallelism, then the system based on Paged Binding Arrays (resp. Stack copying) will behave *exactly* like Aurora (resp. Muse). If a program has only independent and-parallelism the two models will behave *exactly* like &-Prolog (except that conditional bindings would be allocated in the binding array in the system based on Paged Binding Arrays). Our approach can also support the extra-logical features of Prolog (such as cuts and side-effects) transparently [GS91], something which doesn't appear to be possible in other independent-and/or parallel models [BK88, GJ89, RK89]. Control in the models is quite simple, due to recomputation of independent goals. Memory management is also relatively simpler. We firmly believe that the approach, in its two versions of Paged Binding Array and stack copying can be implemented very efficiently, and indeed their implementation is scheduled to begin shortly. The implementation techniques described in this paper can be used for even those models that have dependent and-parallelism, such as Prometheus [SK92], and IDIOM (with recomputation) [GY91]. They can also be extended to implement the Extended Andorra Model [W90].

Acknowledgements

Thanks to Vítor Santos Costa for his numerous comments on this paper, and to Raéd Sindaha and Tony Beaumont for answering many questions about Aurora and its schedulers. The research presented in this paper has benefited from discussions with Kish Shen, Khayri Ali, Roland Carlsson, and David H.D. Warren, all of whom we would like to thank. This research was supported by U.K. Science and Engineering Research Council grant GR/F 27420, ESPRIT project PEPMA and CICYT project TIC90-1105-CE.

References

[AK90] K. Ali, R. Karlsson, "The Muse Or-parallel Prolog Model and its Performance," In *Proceedings of the North American Conference on Logic Programming '90*, MIT Press.

[AK91] K. Ali, R. Karlsson, "Full Prolog and Scheduling Or-parallelism in Muse," To appear in *International Journal of Parallel Programming*, 1991.

[BK88] Uri Baron, et. al., "The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results," In *Proceedings of FGCS '88*, Tokyo, pp. 841-850.

[CA88] W. F. Clocksin and H. Alshawi, "A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors," In *New Generation Computing, 5(1988)*, 361-376.

[CC89] S-E. Chang and Y.P. Chiang, "Restricted And-Parallelism Model with Side Effects," Proceedings of North American Conference on Logic Programming, 1989, MIT Press, pp. 350-368.

[CG91] M. Carro, L. Gomez, and M. Hermenegildo, "VISANDOR: A Tool for Visualizing And-/Or-parallelism in Logic Programs," Technical Report, U. of Madrid (UPM), Madrid-Spain, 1991.

[D91] I. Dutra, "Flexible Scheduling in the Andorra-I System," In *Proc. ICLP'91 Workshop on Parallel Logic Prog.*, Springer Verlag, LNCS 569, Dec. 1991.

[G91] G. Gupta, "Paged Binding Array: Environment Representation in And-Or Parallel Prolog," Technical Report TR-91-24, Department of Computer Science, University of Bristol, Oct. 1991.

[G91a] G. Gupta, "And-Or Parallel Execution of Logic Programs on Shared Memory Multiprocessors," Ph.D. Thesis, University of North Carolina at Chapel Hill, Nov. 1991.

[GS92] G. Gupta, V. Santos Costa, "And-Or Parallel Execution of full Prolog based on Paged Binding Arrays," To appear in Proceedings of Parallel Languages and Architectures Europe (PARLE '92), June 1992.

[GH91] G. Gupta and M. Hermenegildo, "ACE: And/Or-parallel Copying-based Execution of Logic Programs," Technical Report TR-91-25, Department of Computer Science, University of Bristol, Oct. 1991. Also in Springer Verlag LNCS 569, Dec. '91.

[GJ89] G. Gupta and B. Jayaraman, "Compiled And-Or Parallel Execution of Logic Programs," In *Proceedings of the North American Conference on Logic Programming '89*, MIT Press, pp. 332-349.

[GJ90] G. Gupta and B. Jayaraman, "On Criteria for Or-Parallel Execution Models of Logic Programs," In *Proceedings of the North Amer-*

ican Conference on Logic Programming '90, MIT Press, pp. 604-623.

[GJ90a] G. Gupta and B. Jayaraman, "Optimizing And-Or Parallel Implementations," In *Proceedings of the North American Conference on Logic Programming '90*, MIT Press, pp. 737-756.

[GS91] G. Gupta, V. Santos-Costa, "Cut and Side Effects in And-Or Parallel Prolog," Technical Report TR-91-26, Department of Computer Science, University of Bristol, Oct. 1991.

[GY91] G. Gupta, V. Santos Costa, R. Yang, M. Hermenegildo, "IDIOM: A Model for Integrating Dependent-and, Independent-and and Or-parallelism," In *Proc. Int'l. Logic Programming Symposium '91*, MIT Press, Oct. 1991.

[H86] M. V. Hermenegildo, "An Abstract Machine for Restricted And Parallel Execution of Logic Programs". *3rd International Conference on Logic Programming*, London, 1986.

[HG90] M. V. Hermenegildo, K.J. Greene, "&-Prolog and its performance: Exploiting Independent And-Parallelism," In *Proceedings of the 7th International Conference on Logic Programming*, 1990, pp. 253-268.

[HN86] M. V. Hermenegildo and R. I. Nasr, "Efficient Implementation of backtracking in AND-parallelism", *3rd International Conference on Logic Programming*, London, 1986.

[HC87] B. Hausman, et. al., "Or-Parallel Prolog Made Efficient on Shared Memory Multiprocessors," in *1987 IEEE Int. Symp. in Logic Prog.*, San Francisco, CA.

[HC88] B. Hausman, A. Ciepielewski, and A. Calderwood, "Cut and Side-Effects in Or-Parallel Prolog," In *International Conference on Fifth Generation Computer Systems*, Tokyo, Nov. 88, pp. 831-840.

[LK88] Y-J. Lin and V. Kumar, "AND-parallel execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results", in *Fifth International Logic Programming Conference*, Seattle, WA.

[LW90] E. Lusk, D.H.D. Warren, S. Haridi et. al. "The Aurora Or-Prolog System", In *New Generation Computing*, Vol. 7, No. 2,3, 1990 pp. 243-273.

[MH89] K. Muthukumar and M. Hermenegildo, "Complete and Efficient Methods for Supporting Side-effects in Independent/Restricted And-Parallelism," In *Proc. of ICLP*, 1989.

[MH89a] K. Muthukumar, M. V. Hermenegildo, "Determination of Variable Dependence Information through Abstract Interpretation," In *Proc. of NACLP '89*, MIT Press.

[RS87] M. Ratcliffe, J-C Syre, "A Parallel Logic Programming Language for PEPSys" In *Proceedings of IJCAI '87*, Milan, pp. 48-55.

[RK89] B. Ramkumar and L. V. Kalé, "Compiled Execution of the REDUCE-OR Process Model," In *Proc. of NACLP '89*, MIT Press, pp. 313-331.

[S91] R. Sindaha, "The Dharma Scheduler — Definitive Scheduling in Aurora on Multiprocessor Architecture," Technical Report, Department of Computer Science, University of Bristol, forthcoming.

[S89] P. Szeredi, "Performance Analysis of the Aurora Or-parallel Prolog System," In *Proc. of NACLP*, MIT Press, 1989, pp. 713-732.

[SH91] K. Shen and M. Hermenegildo, "A Simulation Study of Or- and Independent And-Parallelism," In *Proc. Int'l. Logic Programming Symposium '91*, MIT Press, Oct. 1991.

[SI91] R. Sindaha, Personal Communication, Sep. 1991.

[SK92] K. Shen, "Studies of And-Or Parallelism in Prolog," Ph.D. thesis, Cambridge University, 1992, forthcoming.

[SW91] V. Santos Costa, D. H. D. Warren, R. Yang, "Andorra-I: A Parallel Prolog system that transparently exploits both And- and Or-Parallelism," In *Proceedings of Principles & Practice of Parallel Programming*, Apr. '91, pp. 83-93.

[VX91] A. Véron, J. Xu, et. al., "Virtual Memory Support for Parallel Logic Programming Systems," In *PARLE'91*, Springer Verlag, LNCS 506, 1991.

[W84] D. S. Warren, "Efficient Prolog Memory Management for Flexible Control Strategies," In *The 1984 Int. Symp. on Logic Prog.*, Atlantic City, pp. 198-202.

[W87] D. H. D. Warren, "The SRI-model for Or-Parallel execution of Prolog – Abstract Design and Implementation Issues," *1987 IEEE Int. Symp. in Logic Prog.*, San Francisco.

[W90] D.H.D. Warren, "Extended Andorra Model with Implicit Control" Talk given at Workshop on Parallel Logic Programming, 7th ICLP, Eilat.